MLApronIDL: OCaml interface for APRON library

Bertrand Jeannet

December 19, 2006

Contents

Chapter 1

Introduction

This package is an OCaml interface for the APRON library/interface. The interface is accessed via the module Apron, which is decomposed into 15 submodules, corresponding to C modules:

Scalar	:	scalars (numbers)
Interval	:	intervals on scalars
Coeff	:	coefficients (either scalars or intervals)
Dimension	:	dimensions and related operations
Linexpr0	:	(interval) linear expressions, level 0
Lincons0	:	(interval) linear constraints, level 0
Generator0	:	generators, level 0
Manager	:	managers
Abstract0	:	abstract values, level 0
Var	:	variables
Environment	:	environment binding variables to dimensions
Linexpr1	:	(interval) linear expressions, level 1
Lincons1	:	interval) linear constraints, level 1
Generator1	:	generators, level 1
Abstract1	:	abstract values, level 1
Parser	:	strings parsing

Requirements

- M4 preprocessor (standard on any UNIX system)
- APRON library
- GMP library (tested with version 4.0 and up)
- mlgmpidl package
- OCaml 3.0 or up (tested with 3.09)
- Camlidl (tested with 1.05)

Installation

Library Set the file ../Makefile.config to your own setting. You might also have to modify the Makefile for executables

type 'make', and then 'make install'

The OCaml part of the library is named apron.cma (.cmxa, .a) The C part of the library, which is automatically referenced by apron.cma/apron.cmxa, is named libapron_caml.a (libapron_caml_debug.a)

'make install' installs not only .mli, .cmi, but also .idl files.

Documentation The documentation (currently very sketchy) is generated with ocamldoc.

'make mlapronidl.pdf'

'make html' (put the HTML files in the html subdirectoy)

Miscellaneous 'make clean' and 'make distclean' have the usual behaviour.

Compilation and Linking

To make things clearer, we assume an example file `example.ml` which uses both NewPolka (convex polyhedra) and Itv (intervals) libraries, in their versions where integers (resp. rationals) are GMP integers (resp. rationals). We assume that C and OCaml interface and library files are located in directory $APRON/lib.
The native-code compilation command looks like

```
ocamlopt -I $APRON/lib -o exampleg.opt bigarray.cmxa gmp.cmxa apron.cmxa itv.cmxa
polka.cmxa -cclib "-lpolkag -litvmpq"
```

Comments:

1. You need at least the libraries `bigarray` (standard OCaml distribution), `gmp`, and `apron` (standard APRON distribution), plus the one implementing an effective abstract domains: here, `itv`, and `polka`.
2. The C libraries associated to those OCaml libraries (*e.g.,* `gmp_caml`, `itv_caml`, ...) are automatically looked for, with the exception of the libraries implementing abstract domains (*e.g.,* `polkag`, `itvmpq`). The reason is that most of these libraries may be compiled with different internal representations and/or options, while their interface remains the same.

The byte-code compilation process looks like

```
ocamlc -I $APRON/lib -make-runtime -o myrun bigarray.cma gmp.cma apron.cma itv.cma
polka.cma -cclib "-lpolkag -litvmpq"

ocamlc -I $APRON/lib -use-runtime myrun -o exampleg bigarray.cma gmp.cma apron.cma
itv.cma polka.cma -cclib "-lpolkag -litvmpq" example.ml
```

Comments:

1. One first build a custom bytecode interpreter that includes the new native-code needed;
2. One then compile the `example.ml` file.

The automatic search for C libraries associated to these OCaml libraries can be disabled by the option `-noautolink` supported by both `ocamlc` and `ocamlopt` commands. For instance, the command for native-code compilation can alternatively looks like:

```
ocamlopt -I $APRON/lib -noautolink -o exampleg.opt bigarray.cmxa gmp.cmxa apron.cmxa
itv.cmxa polka.cmxa -cclib "-lpolkag -polka_caml -litvmpq -litv_caml -lapron_caml
-lapron -lgmp_caml -LGMP/lib -lgmp"
```

The option `-verbose` helps to understand what is happening in case of problem.

Part I

Coefficients

Chapter 2

Module Mpzf : GMP multi-precision integers, functional version

Functions in this module has a functional semantics, unlike the corresponding functions in Mpz[23]. These functions are less efficients, due to the additional memory allocation neded for the result.

`type t`

multi-precision integer

`val to_mpz : t -> Mpz.t`

`val of_mpz : Mpz.t -> t`

Conversion from and to Mpz.t.

There is no sharing between the argument and the result.

2.1 Constructors

```
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
```

2.2 Conversions and Printing

```
val to_string : t -> string
val to_float : t -> float
val print : Format.formatter -> t -> unit
```

2.3 Arithmetic Functions

```
val add : t -> t -> t
val add_int : t -> int -> t
val sub : t -> t -> t
val sub_int : t -> int -> t
val mul : t -> t -> t
val mul_int : t -> int -> t
```

```
val cdiv_q : t -> t -> t
val cdiv_r : t -> t -> t
val cdiv_qr : t -> t -> t * t
val fdiv_q : t -> t -> t
val fdiv_r : t -> t -> t
val fdiv_qr : t -> t -> t * t
val tdiv_q : t -> t -> t
val tdiv_r : t -> t -> t
val tdiv_qr : t -> t -> t * t
val divexact : t -> t -> t
val gmod : t -> t -> t
val gcd : t -> t -> t
val lcm : t -> t -> t
val neg : t -> t
val abs : t -> t
```

2.4 Comparison Functions

```
val cmp : t -> t -> int
val cmp_int : t -> int -> int
val sgn : t -> int
```

Chapter 3

Module Mpqf : GMP multi-precision rationals, functional version

Functions in this module has a functional semantics, unlike the corresponding functions in Mpq[24]. These functions are less efficients, due to the additional memory allocation neded for the result.

```
type t
```

multi-precision rationals

```
val to_mpq : t -> Mpq.t
val of_mpq : Mpq.t -> t
val of_mpz : Mpz.t -> t
val of_mpz2 : Mpz.t -> Mpz.t -> t
```

Conversion from and to Mpz.t and Mpz.t.

There is no sharing between the argument and the result.

3.1 Constructors

```
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
val of_frac : int -> int -> t
val of_mpzf : Mpzf.t -> t
val of_mpzf2 : Mpzf.t -> Mpzf.t -> t
```

3.2 Conversions and Printing

```
val to_string : t -> string
val to_float : t -> float
val print : Format.formatter -> t -> unit
```

3.3 Arithmetic Functions

```
val add : t -> t -> t
```

```
val sub : t -> t -> t
val mul : t -> t -> t
val div : t -> t -> t
val neg : t -> t
val abs : t -> t
val inv : t -> t
val equal : t -> t -> bool
```

3.4 Comparison Functions

```
val cmp : t -> t -> int
val cmp_int : t -> int -> int
val cmp_frac : t -> int -> int -> int
val sgn : t -> int
```

3.5 Extraction Functions

```
val get_num : t -> Mpzf.t
val get_den : t -> Mpzf.t
```

Chapter 4

Module Scalar : APRON Scalar numbers.

See Mpqf[3] for operations on GMP multiprecision rational numbers.

```
type t =
  | Float of float
  | Mpqf of Mpqf.t
```

APRON Scalar numbers. See Mpqf[3] for operations on GMP multiprecision rational numbers.

`val of_mpq : Mpq.t -> t`

`val of_mpqf : Mpqf.t -> t`

`val of_int : int -> t`

`val of_frac : int -> int -> t`

Create a scalar of type Mpqf from resp.

- A multi-precision rational Mpq.t
- A multi-precision rational Mpqf.t
- an integer
- a fraction x/y

`val of_float : float -> t`

Create a scalar of type Float with the given value

`val of_infty : int -> t`

Create a scalar of type Float with the value multiplied by infinity (resulting in minus infinity, zero, or infinity

`val is_infty : t -> int`

Infinity test. is_infty x returns -1 if x is -oo, 1 if x is +oo, and 0 if x is finite.

`val sgn : t -> int`

Return the sign of the coefficient, which may be a negative value, zero or a positive value.

`val cmp : t -> t -> int`

Compare two coefficients, possibly converting one from float to Mpqf.t. compare x y returns a negative number if x is less than y, 0 if they ar equal, and a positive number if x is greater than y.

`val cmp_int : t -> int -> int`

Compare a coefficient with an integer

`val equal : t -> t -> bool`

Equality test, possibly using a conversion from `float` to `Mpqf.t`. Return `true` if the 2 values are equal. Two infinite values of the same signs are considered as equal.

`val equal_int : t -> int -> bool`

Equality test with an integer

`val neg : t -> t`

Negation

`val to_string : t -> string`

Conversion to string, using either `string_of_double` or `Mpqf.to_string`

`val print : Format.formatter -> t -> unit`

Print a coefficient

Chapter 5

Module `Interval` : APRON Intervals on scalars

```
type t = {
  mutable inf : Scalar.t ;
  mutable sup : Scalar.t ;
}
```

APRON Intervals on scalars

`val of_scalar : Scalar.t -> Scalar.t -> t`

Build an interval from a lower and an upper bound

`val of_infsup : Scalar.t -> Scalar.t -> t`

depreciated

`val of_mpq : Mpq.t -> Mpq.t -> t`

`val of_mpqf : Mpqf.t -> Mpqf.t -> t`

`val of_int : int -> int -> t`

`val of_frac : int -> int -> int -> int -> t`

`val of_float : float -> float -> t`

Create an interval from resp. two

- multi-precision rationals `Mpq.t`
- multi-precision rationals `Mpqf.t`
- integers
- fractions `x/y` and `z/w`
- floats

`val is_top : t -> bool`

Does the interval represent the universe (`[-oo,+oo]`) ?

`val is_bottom : t -> bool`

Does the interval contain no value (`[a,b]` with a>b) ?

`val is_leq : t -> t -> bool`

Inclusion test. `is_leq x y` returns `true` if `x` is included in `y`

`val cmp : t -> t -> int`

Non Total Comparison: 0: equality -1: i1 included in i2 +1: i2 included in i1 -2: i1.inf less than or equal to i2.inf +2: i1.inf greater than i2.inf

`val equal : t -> t -> bool`

Equality test

`val is_zero : t -> bool`

Is the coefficient equal to scalar 0 or interval `0,0` ?

`val neg : t -> t`

Negation

`val top : t`

`val bottom : t`

Top and bottom intervals (using `DOUBLE` coefficients)

`val set_infsup : t -> Scalar.t -> Scalar.t -> unit`

Fill the interval with the given lower and upper bouunds

`val set_top : t -> unit`

`val set_bottom : t -> unit`

Fill the interval with top (resp. bottom) value

`val print : Format.formatter -> t -> unit`

Print an interval, under the format `[inf,sup]`

Chapter 6

Module `Coeff` : APRON Coefficients (either scalars or intervals)

```
type union_5 =
  | Scalar of Scalar.t
  | Interval of Interval.t
```

```
type t = union_5
```

APRON Coefficients (either scalars or intervals)

```
val s_of_mpq : Mpq.t -> t
```

```
val s_of_mpqf : Mpqf.t -> t
```

```
val s_of_int : int -> t
```

```
val s_of_frac : int -> int -> t
```

Create a scalar coefficient of type `Mpqf.t` from resp.

- A multi-precision rational `Mpq.t`
- A multi-precision rational `Mpqf.t`
- an integer
- a fraction `x/y`

```
val s_of_float : float -> t
```

Create an interval coefficient of type `Float` with the given value

```
val i_of_scalar : Scalar.t -> Scalar.t -> t
```

Build an interval from a lower and an upper bound

```
val i_of_mpq : Mpq.t -> Mpq.t -> t
```

```
val i_of_mpqf : Mpqf.t -> Mpqf.t -> t
```

```
val i_of_int : int -> int -> t
```

```
val i_of_frac : int -> int -> int -> int -> t
```

```
val i_of_float : float -> float -> t
```

Create an interval coefficient from resp. two

- multi-precision rationals `Mpq.t`
- multi-precision rationals `Mpqf.t`
- integers

- fractions `x/y` and `z/w`
- floats

```
val is_scalar : t -> bool
```

```
val is_interval : t -> bool
```

```
val cmp : t -> t -> int
```

Non Total Comparison:

- If the 2 coefficients are both scalars, corresp. to Scalar.cmp
- If the 2 coefficients are both intervals, corresp. to Interval.cmp
- otherwise, -3 if the first is a scalar, 3 otherwise

```
val equal : t -> t -> bool
```

Equality test

```
val is_zero : t -> bool
```

Is the coefficient equal to scalar 0 or interval `0,0` ?

```
val neg : t -> t
```

Negation

```
val reduce : t -> t
```

Convert interval to scalar if possible

```
val print : Format.formatter -> t -> unit
```

Printing

Part II

Managers and Abstract Domains

Chapter 7

Module Manager : APRON Managers

```
type tbool =
  | False
  | True
  | Top
type funid =
  | Funid_unknown
  | Funid_copy
  | Funid_free
  | Funid_asize
  | Funid_minimize
  | Funid_canonicalize
  | Funid_approximate
  | Funid_is_minimal
  | Funid_is_canonical
  | Funid_fprint
  | Funid_fprintdiff
  | Funid_fdump
  | Funid_serialize_raw
  | Funid_deserialize_raw
  | Funid_bottom
  | Funid_top
  | Funid_of_box
  | Funid_of_lincons_array
  | Funid_dimension
  | Funid_is_bottom
  | Funid_is_top
  | Funid_is_leq
  | Funid_is_eq
  | Funid_is_dimension_unconstrained
  | Funid_sat_interval
  | Funid_sat_lincons
  | Funid_bound_dimension
  | Funid_bound_linexpr
  | Funid_to_box
  | Funid_to_lincons_array
  | Funid_to_generator_array
  | Funid_meet
  | Funid_meet_array
  | Funid_meet_lincons_array
```

```
  | Funid_join
  | Funid_join_array
  | Funid_add_ray_array
  | Funid_assign_linexpr
  | Funid_assign_linexpr_array
  | Funid_substitute_linexpr
  | Funid_substitute_linexpr_array
  | Funid_add_dimensions
  | Funid_remove_dimensions
  | Funid_permute_dimensions
  | Funid_forget_array
  | Funid_expand
  | Funid_fold
  | Funid_widening
  | Funid_closure
  | Funid_size
  | Funid_change_environment
  | Funid_rename_array
  | Funid_size2
type exc =
  | Exc_none
  | Exc_timeout
  | Exc_out_of_space
  | Exc_overflow
  | Exc_invalid_argument
  | Exc_not_implemented
  | Exc_size
type funopt = {
  algorithm : int ;
  approx_before : int ;
  approx_after : int ;
  timeout : int ;
  max_object_size : int ;
  flag_exact_wanted : bool ;
  flag_best_wanted : bool ;
}
type exclog = {
  exn : exc ;
  funid : funid ;
  msg : string ;
}
type t
```

APRON Managers

```
val get_library : t -> string
```

Get the name of the effective library which allocated the manager

```
val get_version : t -> string
```

Get the version of the effective library which allocated the manager

```
val funopt_make : unit -> funopt
```

Return the default options for any function (`0` or `false` for al fields)

```
val get_funopt : t -> funid -> funopt
```

Get the options sets for the function. The result is a copy of the internal structure and may be freely modified

```
val set_funopt : t -> funid -> funopt -> unit
```

Set the options for the function

```
val get_flag_exact : t -> tbool
```

Get the corresponding result flags

```
val get_flag_best : t -> tbool
exception Error of exclog
```

Exception raised by functions of the interface

```
val string_of_tbool : tbool -> string
val string_of_funid : funid -> string
val string_of_exc : exc -> string
val print_tbool : Format.formatter -> tbool -> unit
```

Set / get the global manager used for deserialization

```
val set_deserialize : t -> unit
val get_deserialize : unit -> t
```

Chapter 8

Module `Itv` : ITV: intervals abstract domain

`val manager_alloc : unit -> Apron.Manager.t`

Create an Itv manager.

Chapter 9

Module `Oct` : OCT: octagon abstract domain.

```
type internal
```

```
val manager_alloc : unit -> Apron.Manager.t
```

OCT: octagon abstract domain.

Allocate a new manager to manipulate octagons.

```
val manager_get_internal : Apron.Manager.t -> internal
```

No internal parameters for now...

```
val of_generator_array :
  Apron.Manager.t ->
  int -> int -> Apron.Generator0.t array -> Apron.Abstract0.t
```

Approximate a set of generators to an abstract value, with best precision.

```
val widening_thresholds :
  Apron.Manager.t ->
  Apron.Abstract0.t ->
  Apron.Abstract0.t -> Apron.Scalar.t array -> Apron.Abstract0.t
```

Widening with scalar thresholds.

```
val narrowing :
  Apron.Manager.t ->
  Apron.Abstract0.t -> Apron.Abstract0.t -> Apron.Abstract0.t
```

Standard narrowing.

```
val add_epsilon :
  Apron.Manager.t -> Apron.Abstract0.t -> Apron.Scalar.t -> Apron.Abstract0.t
```

Perturbation.

```
val add_epsilon_bin :
  Apron.Manager.t ->
  Apron.Abstract0.t -> Apron.Abstract0.t -> Apron.Scalar.t -> Apron.Abstract0.t
```

Perturbation.

```
val pre_widening : int
```

Algorithms.

Chapter 10

Module `Polka` : POLKA: Convex Polyhedra abstract domain

```
type internal
```

POLKA: Convex Polyhedra abstract domain

```
val manager_alloc : bool -> Apron.Manager.t
```

Create a NewPolka manager. A true Boolean enables strict constraints.

```
val manager_get_internal : Apron.Manager.t -> internal
```

Get the internal submanager of a NewPolka manager.

Various options. See the C documentation

```
val set_max_coeff_size : internal -> int -> unit
val set_approximate_max_coeff_size : internal -> int -> unit
val get_max_coeff_size : internal -> int
val get_approximate_max_coeff_size : internal -> int
```

Part III

Level 1 of the interface

Chapter 11

Module Var : APRON Variables

```
type t
val of_string : string -> t
val compare : t -> t -> int
val to_string : t -> string
val hash : t -> int
```

APRON Variables

```
val print : Format.formatter -> t -> unit
```

Chapter 12

Module Environment : APRON Environments binding dimensions to names

```
type typvar =
  | INT
  | REAL
```

```
type t
```

APRON Environments binding dimensions to names

```
val make : Var.t array -> Var.t array -> t
```

Making an environment from a set of integer and real variables. Raise `Failure` in case of name conflict.

```
val add : t -> Var.t array -> Var.t array -> t
```

Adding to an environment a set of integer and real variables. Raise `Failure` in case of name conflict.

```
val remove : t -> Var.t array -> Var.t array -> t
```

Remove from an environment a set of integer and real variables. Raise `Failure` in case of non-existing variables.

```
val equal : t -> t -> bool
```

Test equality if two environments

```
val compare : t -> t -> int
```

Compare two environment. `compare env1 env2` return `-2` if the environements are not compatible (a variable has different types in the 2 environements), `-1` if `env1` is a subset of env2, `0` if equality, `+1` if env1 is a superset of env2, and `+2` otherwise (the lce exists and is a strict superset of both)

```
val dimension : t -> Dim.dimension
```

Return the dimension of the environment

```
val size : t -> int
```

Return the size of the environment

```
val typ_of_var : t -> Var.t -> typvar
```

Return the type of variables in the environment. If the variable does not belong to the environment, raise a `Failure` exception.

```
val vars : t -> Var.t array * Var.t array
```

Return the (lexicographically ordered) sets of integer and real variables in the environment

```
val var_of_dim : t -> Dim.t -> Var.t
```

Return the variable corresponding to the given dimension in the environment. Raise `Failure` is the dimension is out of the range of the environment (greater than or equal to `dim env`)

```
val dim_of_var : t -> Var.t -> Dim.t
```

Return the dimension associated to the given variable in the environment. Raise `Failure` if the variable does not belong to the environment.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> t -> unit
```

Printing

Chapter 13

Module Linexpr1 : APRON Expressions of level 1

```
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable env : Environment.t ;
}
```

APRON Expressions of level 1

`val make : ?sparse:bool -> Environment.t -> t`

Build a linear expression defined on the given argument, which is sparse by default.

`val minimize : t -> unit`

In case of sparse representation, remove zero coefficients

`val copy : t -> t`

Copy

`val print : Format.formatter -> t -> unit`

Print the linear expression

`val set_list : t -> (Coeff.t * Var.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")] (Some cst)` assigns coefficients `c1` to variable `"x"`, coefficient `c2` to variable `"y"`, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

`val set_array : t -> (Coeff.t * Var.t) array -> Coeff.t option -> unit`

Set simultaneously a number of coefficients, as `set_list`.

`val iter : (Coeff.t -> Var.t -> unit) -> t -> unit`

Iter the function on the pair coefficient/variable of the linear expression

`val get_cst : t -> Coeff.t`

Get the constant

`val set_cst : t -> Coeff.t -> unit`

Set the constant

`val get_coeff : t -> Var.t -> Coeff.t`

Get the coefficient of the variable

`val set_coeff : t -> Var.t -> Coeff.t -> unit`

Set the coefficient of the variable

`val extend_environment : t -> Environment.t -> t`

Change the environment of the expression for a super-environement. Raise `Failure` if it is not the case

`val extend_environment_with : t -> Environment.t -> unit`

Side-effet version of the previous function

`val is_integer : t -> bool`

Does the linear expression depend only on integer variables ?

`val is_real : t -> bool`

Does the linear expression depend only on real variables ?

`val get_linexpr0 : t -> Linexpr0.t`

Get the underlying expression of level 0 (which is not a copy).

`val get_env : t -> Environment.t`

Get the environement of the expression

Chapter 14

Module Lincons1 : APRON Constraints and array of constraints of level 1

```
type t = {
  mutable lincons0 : Lincons0.t ;
  mutable env : Environment.t ;
}
type earray = {
  mutable lincons0_array : Lincons0.t array ;
  mutable array_env : Environment.t ;
}
```

APRON Constraints and array of constraints of level 1

```
type typ = Lincons0.typ =
  | EQ
  | SUPEQ
  | SUP
```

```
val make : Linexpr1.t -> Lincons0.typ -> t
```

Make a linear constraint. Modifying later the linear expression (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

```
val copy : t -> t
```

Copy (deep copy)

```
val print : Format.formatter -> t -> unit
```

Print the linear constraint

```
val get_typ : t -> Lincons0.typ
```

Get the constraint type

```
val iter : (Coeff.t -> Var.t -> unit) -> t -> unit
```

Iter the function on the pair coefficient/variable of the underlying linear expression

```
val get_cst : t -> Coeff.t
```

Get the constant of the underlying linear expression

```
val set_typ : t -> Lincons0.typ -> unit
```

Set the constraint type

`val set_list : t -> (Coeff.t * Var.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")] (Some cst)` assigns coefficients `c1` to variable `"x"`, coefficient `c2` to variable `"y"`, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

`val set_array : t -> (Coeff.t * Var.t) array -> Coeff.t option -> unit`

Set simultaneously a number of coefficients, as `set_list`.

`val set_cst : t -> Coeff.t -> unit`

Set the constant of the underlying linear expression

`val get_coeff : t -> Var.t -> Coeff.t`

Get the coefficient of the variable in the underlying linear expression

`val set_coeff : t -> Var.t -> Coeff.t -> unit`

Set the coefficient of the variable in the underlying linear expression

`val make_unsat : Environment.t -> t`

Build the unsatisfiable constraint -1≥0

`val is_unsat : t -> bool`

Is the constraint not satisfiable ?

`val extend_environment : t -> Environment.t -> t`

Change the environement of the constraint for a super-environement. Raise `Failure` if it is not the case

`val extend_environment_with : t -> Environment.t -> unit`

Side-effect version of the previous function

`val get_env : t -> Environment.t`

Get the environement of the linear constraint

`val get_linexpr1 : t -> Linexpr1.t`

Get the underlying linear expression. Modifying the linear expression (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

`val get_lincons0 : t -> Lincons0.t`

Get the underlying linear constraint of level 0. Modifying the constraint of level 0 (*not advisable*) modifies correspondingly the linear constraint and conversely, except for changes of environements

14.1 Type array

`val array_make : Environment.t -> int -> earray`

Make an array of linear constraints with the given size and defined on the given environement. The elements are initialized with the constraint 0=0.

```
val array_print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> earray -> unit
```

Print an array of constraints

```
val array_length : earray -> int
```

Get the size of the array

```
val array_get_env : earray -> Environment.t
```

Get the environment of the array

```
val array_get : earray -> int -> t
```

Get the element of the given index (which is not a copy)

```
val array_set : earray -> int -> t -> unit
```

Set the element of the given index (without any copy). The array and the constraint should be defined on the same environement; otherwise a `Failure` exception is raised.

```
val array_extend_environment : earray -> Environment.t -> earray
```

Change the environement of the array of constraints for a super-environement. Raise `Failure` if it is not the case

```
val array_extend_environment_with : earray -> Environment.t -> unit
```

Side-effect version of the previous function

Chapter 15

Module `Generator1` : APRON Generators and array of generators of level 1

```
type t = {
  mutable generator0 : Generator0.t ;
  mutable env : Environment.t ;
}
type earray = {
  mutable generator0_array : Generator0.t array ;
  mutable array_env : Environment.t ;
}
```

APRON Generators and array of generators of level 1

```
type typ = Generator0.typ =
  | LINE
  | RAY
  | VERTEX
```

```
val make : Linexpr1.t -> Generator0.typ -> t
```

Make a generator. Modifying later the linear expression (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

```
val copy : t -> t
```

Copy (deep copy)

```
val print : Format.formatter -> t -> unit
```

Print the generator

```
val get_typ : t -> Generator0.typ
```

Get the generator type

```
val iter : (Coeff.t -> Var.t -> unit) -> t -> unit
```

Iter the function on the pair coefficient/variable of the underlying linear expression

```
val set_typ : t -> Generator0.typ -> unit
```

Set the generator type

```
val set_list : t -> (Coeff.t * Var.t) list -> unit
```

Set simultaneously a number of coefficients.

`set_list expr [(c1,"x"); (c2,"y")]` assigns coefficients `c1` to variable `"x"` and coefficient `c2` to variable `"y"`.

```
val set_array : t -> (Coeff.t * Var.t) array -> unit
```

Set simultaneously a number of coefficients, as `set_list`.

```
val get_coeff : t -> Var.t -> Coeff.t
```

Get the coefficient of the variable in the underlying linear expression

```
val set_coeff : t -> Var.t -> Coeff.t -> unit
```

Set the coefficient of the variable in the underlying linear expression

```
val extend_environment : t -> Environment.t -> t
```

Change the environement of the generator for a super-environement. Raise `Failure` if it is not the case

```
val extend_environment_with : t -> Environment.t -> unit
```

Side-effect version of the previous function

15.1 Type earray

```
val array_make : Environment.t -> int -> earray
```

Make an array of generators with the given size and defined on the given environement. The elements are initialized with the line 0.

```
val array_print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  Format.formatter -> earray -> unit
```

Print an array of generators

```
val array_length : earray -> int
```

Get the size of the array

```
val array_get : earray -> int -> t
```

Get the element of the given index (which is not a copy)

```
val array_set : earray -> int -> t -> unit
```

Set the element of the given index (without any copy). The array and the generator should be defined on the same environement; otherwise a `Failure` exception is raised.

```
val array_extend_environment : earray -> Environment.t -> earray
```

Change the environement of the array of generators for a super-environement. Raise `Failure` if it is not the case

```
val array_extend_environment_with : earray -> Environment.t -> unit
```

Side-effect version of the previous function

```
val get_env : t -> Environment.t
```

Get the environement of the generator

```
val get_linexpr1 : t -> Linexpr1.t
```

Get the underlying linear expression. Modifying the linear expression (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

```
val get_generator0 : t -> Generator0.t
```

Get the underlying generator of level 0. Modifying the generator of level 0 (*not advisable*) modifies correspondingly the generator and conversely, except for changes of environements

Chapter 16

Module Abstract1 : APRON Abstract values of level 1

```
type t = {
  mutable abstract0 : Abstract0.t ;
  mutable env : Environment.t ;
}
type box1 = {
  mutable interval_array : Interval.t array ;
  mutable box1_env : Environment.t ;
}
```

APRON Abstract values of level 1

16.1 General management

16.1.1 Memory

`val copy : Manager.t -> t -> t`

Copy a value

`val size : Manager.t -> t -> int`

Return the abstract size of a value

16.1.2 Control of internal representation

`val minimize : Manager.t -> t -> unit`

Minimize the size of the representation of the value. This may result in a later recomputation of internal information.

`val canonicalize : Manager.t -> t -> unit`

Put the abstract value in canonical form. (not yet clear definition)

`val approximate : Manager.t -> t -> 'a -> int -> unit`

`approximate man abs alg` perform some transformation on the abstract value, guided by the argument `alg`. The transformation may lose information. The argument `alg` overrides the field algorithm of the structure of type `Manager.funopt` associated to ap_abstract0_approximate (commodity feature).

```
val is_minimal : Manager.t -> t -> Manager.tbool
```

Is the value in minimal form ?

```
val is_canonical : Manager.t -> t -> Manager.tbool
```

Is the value in canonical from ?

16.1.3 Printing

```
val fdump : Manager.t -> t -> unit
```

Dump on the `stdout` C stream the internal representation of an abstract value, for debugging purposes

```
val print : Format.formatter -> t -> unit
```

Print as a set of constraints

16.1.4 Serialization

16.2 Constructor, accessors, tests and property extraction

16.2.1 Basic constructors

All these functions request explicitly an environment in their arguments.

```
val bottom : Manager.t -> Environment.t -> t
```

Create a bottom (empty) value defined on the given environment

```
val top : Manager.t -> Environment.t -> t
```

Create a top (universe) value defined on the given environment

```
val of_box :
  Manager.t -> Environment.t -> Var.t array -> Interval.t array -> t
```

Abstract an hypercube.

`of_box man env tvar tinterval` abstracts an hypercube defined by the arrays `tvar` and `tinterval`. The result is defined on the environment `env`, which should contain all the variables in `tvar` (and defines their type)

```
val of_lincons_array : Manager.t -> Environment.t -> Lincons1.earray -> t
```

Abstract a convex polyhedron.

`of_lincons_array man env array` abstracts the convex polyhedron defined by the array of constraints with an abstract value defined on the environment `env`.

16.2.2 Accessors

```
val manager : t -> Manager.t
val env : t -> Environment.t
val abstract0 : t -> Abstract0.t
```

Return resp. the underlying manager, environment and abstract value of level 0

16.2.3 Tests

`val is_bottom : Manager.t -> t -> Manager.tbool`

Emptiness test

`val is_top : Manager.t -> t -> Manager.tbool`

Universality test

`val is_leq : Manager.t -> t -> t -> Manager.tbool`

Inclusion test. The 2 abstract values should be compatible.

`val is_eq : Manager.t -> t -> t -> Manager.tbool`

Equality test. The 2 abstract values should be compatible.

`val sat_lincons : Manager.t -> t -> Lincons1.t -> Manager.tbool`

Does the abstract value satisfy the linear constraint ?

`val sat_interval : Manager.t -> t -> Var.t -> Interval.t -> Manager.tbool`

Does the abstract value satisfy the constraint `dim in interval` ?

`val is_variable_unconstrained : Manager.t -> t -> Var.t -> Manager.tbool`

Is the variable unconstrained in the abstract value ? If yes, this means that the existential quantification of the dimension does not change the value.

16.2.4 Extraction of properties

`val bound_variable : Manager.t -> t -> Var.t -> Interval.t`

Return the interval of variation of the variable in the abstract value.

`val bound_linexpr : Manager.t -> t -> Linexpr1.t -> Interval.t`

Return the interval of variation of the linear expression in the abstract value.

Implement a form of linear programming, where the argument linear expression is the one to optimize under the constraints induced by the abstract value.

`val to_box : Manager.t -> t -> box1`

Convert the abstract value to an hypercube

`val to_lincons_array : Manager.t -> t -> Lincons1.earray`

Convert the abstract value to a conjunction of linear constraints.

`val to_generator_array : Manager.t -> t -> Generator1.earray`

Convert the abstract value to a set of generators that defines it.

16.3 Operations

16.3.1 Meet and Join

`val meet : Manager.t -> t -> t -> t`

Meet of 2 abstract values.

```
val meet_array : Manager.t -> t array -> t
```

Meet of a non empty array of abstract values.

```
val meet_lincons_array : Manager.t -> t -> Lincons1.earray -> t
```

Meet of an abstract value with an array of linear constraints.

```
val join : Manager.t -> t -> t -> t
```

Join of 2 abstract values.

```
val join_array : Manager.t -> t array -> t
```

Join of a non empty array of abstract values.

```
val add_ray_array : Manager.t -> t -> Generator1.earray -> t
```

Add the array of generators to the abstract value (time elapse operator).

The generators should either lines or rays, not vertices.

16.3.1.0.1 Side-effect versions of the previous functions

```
val meet_with : Manager.t -> t -> t -> unit
val meet_lincons_array_with : Manager.t -> t -> Lincons1.earray -> unit
val join_with : Manager.t -> t -> t -> unit
val add_ray_array_with : Manager.t -> t -> Generator1.earray -> unit
```

16.3.2 Assignement and Substitutions

```
val assign_linexpr : Manager.t ->
  t -> Var.t -> Linexpr1.t -> t option -> t
```

Assignement of a single dimension by a linear expression

```
val substitute_linexpr :
  Manager.t ->
  t -> Var.t -> Linexpr1.t -> t option -> t
```

Substitution of a single dimension by a linear expression

```
val assign_linexpr_array :
  Manager.t ->
  t ->
  Var.t array -> Linexpr1.t array -> t option -> t
```

Parallel assignement of an array of dimensions by an array of same size of linear expressions

```
val substitute_linexpr_array :
  Manager.t ->
  t ->
  Var.t array -> Linexpr1.t array -> t option -> t
```

Parallel substitution of an array of dimensions by an array of same size of linear expressions

16.3.2.0.2 Side-effect versions of the previous functions

```
val assign_linexpr_with :
  Manager.t -> t -> Var.t -> Linexpr1.t -> t option -> unit
val substitute_linexpr_with :
  Manager.t -> t -> Var.t -> Linexpr1.t -> t option -> unit
val assign_linexpr_array_with :
  Manager.t ->
  t -> Var.t array -> Linexpr1.t array -> t option -> unit
val substitute_linexpr_array_with :
  Manager.t ->
  t -> Var.t array -> Linexpr1.t array -> t option -> unit
```

16.3.3 Projections

These functions implements forgeting (existential quantification) of (array of) variables. Both functional and side-effect versions are provided. The Boolean, if true, adds a projection onto 0-plane.

```
val forget_array : Manager.t -> t -> Var.t array -> bool -> t
val forget_array_with : Manager.t -> t -> Var.t array -> bool -> unit
```

16.3.4 Change and permutation of dimensions

```
val change_environment : Manager.t -> t -> Environment.t -> bool -> t
```

Change the environement of the abstract values.

Variables that are removed are first existentially quantified, and variables that are introduced are unconstrained. The Boolean, if true, adds a projection onto 0-plane for these ones.

```
val minimize_environment : Manager.t -> t -> t
```

Remove from the environment of the abstract value and from the abstract value itself variables that are unconstrained in it.

```
val rename_array : Manager.t -> t -> Var.t array -> Var.t array -> t
```

Parallel renaming of the environment of the abstract value.

The new variables should not interfere with the variables that are not renamed.

```
val change_environment_with : Manager.t -> t -> Environment.t -> bool -> unit
val minimize_environment_with : Manager.t -> t -> unit
val rename_array_with : Manager.t -> t -> Var.t array -> Var.t array -> unit
```

16.3.5 Expansion and folding of dimensions

These functions allows to expand one dimension into several ones having the same properties with respect to the other dimensions, and to fold several dimensions into one. Formally,

- expand P(x,y,z) z w = P(x,y,z) inter P(x,y,w) if z is expanded in z and w
- fold Q(x,y,z,w) z w = exists w:Q(x,y,z,w) union (exist z:Q(x,y,z,w))(z←w) if z and w are folded onto z

```
val expand : Manager.t -> t -> Var.t -> Var.t array -> t
```

Expansion: `expand a var tvar` expands the variable `var` into itself and the additional variables in `tvar`, which are given the same type as `var`.

It results in (n+1) unrelated variables having same relations with other variables. The additional variables are added to the environment of the argument for making the environment of the result, so they should not belong to the initial environement.

```
val fold : Manager.t -> t -> Var.t array -> t
```

Folding: `fold a tvar` fold the variables in the array `tvar` of size n≥1 and put the result in the first variable of the array. The other variables of the array are then removed, both from the environment and the abstract value.

```
val expand_with : Manager.t -> t -> Var.t -> Var.t array -> unit
val fold_with : Manager.t -> t -> Var.t array -> unit
```

16.3.6 Widening

```
val widening : Manager.t -> t -> t -> t
```

Widening

```
val widening_threshold : Manager.t -> t -> t -> Lincons1.earray -> t
```

16.3.7 Closure operation

```
val closure : Manager.t -> t -> t
```

Closure: transform strict constraints into non-strict ones.

```
val closure_with : Manager.t -> t -> unit
```

Side-effect version

Chapter 17

Module Parser : APRON Parsing of expressions

17.1 Introduction

This small module implements the parsing of expressions, constraints and generators. The allowed syntax is simple (no parenthesis) but supports interval expressions.

```
cons ::= expr ('>' | '>=' | '=' | '!=' | '=' | '<=' | '<') expr
gen ::= ('V:' | 'R:' | 'L:') expr
expr ::= expr '+' term | expr '-' term | term
term ::= coeff ['*'] identifier | coeff | ['-'] identifier
coeff ::= scalar | ['-'] '['scalar ';' scalar ']'
scalar ::= ['-'] (integer | rational | floating_point_number)
```

There is the possibility to parse directly from a lexing buffer, or from a string (from which one can generate a buffer with the function `Lexing.from_string`.

This module uses the underlying modules `Apron_lexer` and `Apron_parser`.

17.2 Interface

```
exception Error of string
```

Raised by conversion functions

```
val linexpr1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Linexpr1.t
val lincons1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Lincons1.t
val generator1_of_lexbuf : Environment.t -> Lexing.lexbuf -> Generator1.t
```

Conversion from lexing buffers to resp. linear expressions, linear constraints and generators, defined on the given environment.

```
val linexpr1_of_string : Environment.t -> string -> Linexpr1.t
val lincons1_of_string : Environment.t -> string -> Lincons1.t
val generator1_of_string : Environment.t -> string -> Generator1.t
```

Conversion from strings to resp. linear expressions, linear constraints and generators, defined on the given environment.

```
val lincons1_of_lstring : Environment.t -> string list -> Lincons1.earray
val generator1_of_lstring : Environment.t -> string list -> Generator1.earray
```

Conversion from lists of strings to array of resp. linear constraints and generators, defined on the given environment.

```
val of_lstring : Manager.t -> Environment.t -> string list -> Abstract1.t
```

Abstraction of lists of strings representing constraints to abstract values, on the abstract domain defined by the given manager.

Part IV

Level 0 of the interface

Chapter 18

Module `Dim` : APRON Dimensions and related types

```
type t = int
type change = {
  dim : int array ;
  intdim : int ;
  realdim : int ;
}
type perm = int array
type dimension = {
  intd : int ;
  reald : int ;
}
```

APRON Dimensions and related types

Chapter 19

Module `Linexpr0` : APRON Linear expressions of level 0

`type t`
APRON Linear expressions of level 0

`val make : int option -> t`

Create a linear expression. Its representation is sparse if `None` is provided, dense of size `size` if `Some size` is provided.

`val minimize : t -> unit`

In case of sparse representation, remove zero coefficients

`val copy : t -> t`

Copy

`val compare : t -> t -> int`

Comparison with lexicographic ordering using Coeff.cmp, terminating by constant

`val hash : t -> int`

Hashing function

`val get_size : t -> int`

Get the size of the linear expression (which may be sparse or dense)

`val get_cst : t -> Coeff.t`

Get the constant

`val get_coeff : t -> Dim.t -> Coeff.t`

Get the coefficient corresponding to the dimension

`val set_list : t -> (Coeff.t * Dim.t) list -> Coeff.t option -> unit`

Set simultaneously a number of coefficients.

`set_list expr [(c1,1); (c2,2)] (Some cst)` assigns coefficients `c1` to dimension 1, coefficient `c2` to dimension 2, and coefficient `cst` to the constant. If `(Some cst)` is replaced by `None`, the constant coefficient is not assigned.

`val set_array : t -> (Coeff.t * Dim.t) array -> Coeff.t option -> unit`

Set simultaneously a number of coefficients, as `set_list`.

```
val set_cst : t -> Coeff.t -> unit
```

Set the constant

```
val set_coeff : t -> Dim.t -> Coeff.t -> unit
```

Set the coefficient corresponding to the dimension

Iter the function on the pairs coefficient/dimension of the linear expression

```
val iter : (Coeff.t -> Dim.t -> unit) -> t -> unit
```

```
val print : (Dim.t -> string) -> Format.formatter -> t -> unit
```

Print a linear expression, using a function converting from dimensions to names

Chapter 20

Module Lincons0 : APRON Linear constraints of level 0

```
type typ =
  | EQ
  | SUPEQ
  | SUP
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable typ : typ ;
}
```

APRON Linear constraints of level 0

`val make : Linexpr0.t -> typ -> t`

Make a linear constraint. Modifying later the linear expression modifies correspondingly the linear constraint and conversely

`val copy : t -> t`

Copy a linear expression (deep copy)

`val string_of_typ : typ -> string`

Convert a constraint type to a string (=,>=, or >)

`val print : (Dim.t -> string) -> Format.formatter -> t -> unit`

Print a constraint

Chapter 21

Module Generator0 : APRON Generators of level 0

```
type typ =
  | LINE
  | RAY
  | VERTEX
type t = {
  mutable linexpr0 : Linexpr0.t ;
  mutable typ : typ ;
}
```

APRON Generators of level 0

`val make : Linexpr0.t -> typ -> t`

Making a generator. The constant coefficient of the linear expression is ignored. Modifying later the linear expression modifies correspondingly the generator and conversely.

`val copy : t -> t`

Copy a generator (deep copy)

`val string_of_typ : typ -> string`

Convert a generator type to a string (`LIN`,`RAY`, or `VTX`)

`val print : (Dim.t -> string) -> Format.formatter -> t -> unit`

Print a generator

Chapter 22

Module `Abstract0` : APRON Abstract value of level 0

`type t`
APRON Abstract value of level 0
`val set_gc : int -> unit`
TO BE DOCUMENTED

22.1 General management

22.1.1 Memory

`val copy : Manager.t -> t -> t`
Copy a value

`val size : Manager.t -> t -> int`
Return the abstract size of a value

22.1.2 Control of internal representation

`val minimize : Manager.t -> t -> unit`
Minimize the size of the representation of the value. This may result in a later recomputation of internal information.

`val canonicalize : Manager.t -> t -> unit`
Put the abstract value in canonical form. (not yet clear definition)

`val approximate : Manager.t -> t -> int -> unit`
`approximate man abs alg` perform some transformation on the abstract value, guided by the argument `alg`. The transformation may lose information. The argument `alg` overrides the field algorithm of the structure of type `Manager.funopt` associated to ap_abstract0_approximate (commodity feature).

`val is_minimal : Manager.t -> t -> Manager.tbool`
Is the value in minimal form ?

```
val is_canonical : Manager.t -> t -> Manager.tbool
```

Is the value in canonical from ?

22.1.3 Printing

```
val fdump : Manager.t -> t -> unit
```

Dump on the `stdout` C stream the internal representation of an abstract value, for debugging purposes

```
val print : (int -> string) -> Format.formatter -> t -> unit
```

Print as a set of constraints

22.1.4 Serialization

22.2 Constructor, accessors, tests and property extraction

22.2.1 Basic constructors

```
val bottom : Manager.t -> int -> int -> t
```

Create a bottom (empty) value with the given number of integer and real variables

```
val top : Manager.t -> int -> int -> t
```

Create a top (universe) value with the given number of integer and real variables

```
val of_box : Manager.t -> int -> int -> Interval.t array -> t
```

Abstract an hypercube.

`of_box man intdim realdim array` abstracts an hypercube defined by the array of intervals of size `intdim+realdim`

```
val of_lincons_array : Manager.t -> int -> int -> Lincons0.t array -> t
```

Abstract a convex polyhedron.

`of_lincons_array man intdim realdim array` abstracts the convex polyhedron defined by the array of constraints with an abstract value having the specified number of integer and real dimensions.

22.2.2 Accessors

```
val dimension : Manager.t -> t -> Dim.dimension
val manager : t -> Manager.t
```

22.2.3 Tests

```
val is_bottom : Manager.t -> t -> Manager.tbool
```

Emptiness test

```
val is_top : Manager.t -> t -> Manager.tbool
```

Universality test

```
val is_leq : Manager.t -> t -> t -> Manager.tbool
```

Inclusion test. The 2 abstract values should be compatible.

```
val is_eq : Manager.t -> t -> t -> Manager.tbool
```

Equality test. The 2 abstract values should be compatible.

```
val sat_lincons : Manager.t -> t -> Lincons0.t -> Manager.tbool
```

Does the abstract value satisfy the linear constraint ?

```
val sat_interval : Manager.t -> t -> Dim.t -> Interval.t -> Manager.tbool
```

Does the abstract value satisfy the constraint `dim in interval` ?

```
val is_dimension_unconstrained : Manager.t -> t -> Dim.t -> Manager.tbool
```

Is the dimension unconstrained in the abstract value ? If yes, this means that the existential quantification of the dimension does not change the value.

22.2.4 Extraction of properties

```
val bound_dimension : Manager.t -> t -> Dim.t -> Interval.t
```

Return the interval of variation of the dimension in the abstract value.

```
val bound_linexpr : Manager.t -> t -> Linexpr0.t -> Interval.t
```

Return the interval of variation of the linear expression in the abstract value.

Implement a form of linear programming, where the argument linear expression is the one to optimize under the constraints induced by the abstract value.

```
val to_box : Manager.t -> t -> Interval.t array
```

Convert the abstract value to an hypercube

```
val to_lincons_array : Manager.t -> t -> Lincons0.t array
```

Convert the abstract value to a conjunction of linear constraints.

```
val to_generator_array : Manager.t -> t -> Generator0.t array
```

Convert the abstract value to a set of generators that defines it.

22.3 Operations

22.3.1 Meet and Join

```
val meet : Manager.t -> t -> t -> t
```

Meet of 2 abstract values.

```
val meet_array : Manager.t -> t array -> t
```

Meet of a non empty array of abstract values.

```
val meet_lincons_array : Manager.t -> t -> Lincons0.t array -> t
```

Meet of an abstract value with an array of linear constraints.

```
val join : Manager.t -> t -> t -> t
```

Join of 2 abstract values.

```
val join_array : Manager.t -> t array -> t
```

Join of a non empty array of abstract values.

```
val add_ray_array : Manager.t -> t -> Generator0.t array -> t
```

Add the array of generators to the abstract value (time elapse operator).

The generators should either lines or rays, not vertices.

22.3.1.0.3 Side-effect versions of the previous functions

```
val meet_with : Manager.t -> t -> t -> unit
val meet_lincons_array_with : Manager.t -> t -> Lincons0.t array -> unit
val join_with : Manager.t -> t -> t -> unit
val add_ray_array_with : Manager.t -> t -> Generator0.t array -> unit
```

22.3.2 Assignements and Substitutions

```
val assign_linexpr : Manager.t ->
  t -> Dim.t -> Linexpr0.t -> t option -> t
```

Assignement of a single dimension by a linear expression

```
val substitute_linexpr :
  Manager.t ->
  t -> Dim.t -> Linexpr0.t -> t option -> t
```

Substitution of a single dimension by a linear expression

```
val assign_linexpr_array :
  Manager.t ->
  t ->
  Dim.t array -> Linexpr0.t array -> t option -> t
```

Parallel assignement of an array of dimensions by an array of same size of linear expressions

```
val substitute_linexpr_array :
  Manager.t ->
  t ->
  Dim.t array -> Linexpr0.t array -> t option -> t
```

Parallel substitution of an array of dimensions by an array of same size of linear expressions

22.3.2.0.4 Side-effect versions of the previous functions

```
val assign_linexpr_with :
  Manager.t -> t -> Dim.t -> Linexpr0.t -> t option -> unit
val substitute_linexpr_with :
  Manager.t -> t -> Dim.t -> Linexpr0.t -> t option -> unit
val assign_linexpr_array_with :
  Manager.t ->
  t -> Dim.t array -> Linexpr0.t array -> t option -> unit
val substitute_linexpr_array_with :
  Manager.t ->
  t -> Dim.t array -> Linexpr0.t array -> t option -> unit
```

22.3.3 Projections

These functions implements forgeting (existential quantification) of (array of) dimensions. Both functional and side-effect versions are provided. The Boolean, if true, adds a projection onto 0-plane.

```
val forget_array : Manager.t -> t -> Dim.t array -> bool -> t
val forget_array_with : Manager.t -> t -> Dim.t array -> bool -> unit
```

22.3.4 Change and permutation of dimensions

```
val add_dimensions : Manager.t -> t -> Dim.change -> bool -> t
val remove_dimensions : Manager.t -> t -> Dim.change -> t
val permute_dimensions : Manager.t -> t -> Dim.perm option -> t
```

22.3.4.0.5 Side-effect versions of the previous functions

```
val add_dimensions_with : Manager.t -> t -> Dim.change -> bool -> unit
val remove_dimensions_with : Manager.t -> t -> Dim.change -> unit
val permute_dimensions_with : Manager.t -> t -> Dim.perm option -> unit
```

22.3.5 Expansion and folding of dimensions

These functions allows to expand one dimension into several ones having the same properties with respect to the other dimensions, and to fold several dimensions into one. Formally,

- expand P(x,y,z) z w = P(x,y,z) inter P(x,y,w) if z is expanded in z and w
- fold Q(x,y,z,w) z w = exists w:Q(x,y,z,w) union (exist z:Q(x,y,z,w))(z←w) if z and w are folded onto z

```
val expand : Manager.t -> t -> Dim.t -> int -> t
```

Expansion: `expand a dim n` expands the dimension `dim` into itself + `n` additional dimensions. It results in (n+1) unrelated dimensions having same relations with other dimensions. The (n+1) dimensions are put as follows:

- original dimension `dim`
- if the dimension is integer, the n additional dimensions are put at the end of integer dimensions; if it is real, at the end of the real dimensions.

```
val fold : Manager.t -> t -> Dim.t array -> t
```

Folding: `fold a tdim` fold the dimensions in the array `tdim` of size n≥1 and put the result in the first dimension of the array. The other dimensions of the array are then removed (using ap_abstract0_permute_remove_dimensions).

```
val expand_with : Manager.t -> t -> Dim.t -> int -> unit
val fold_with : Manager.t -> t -> Dim.t array -> unit
```

22.3.6 Widening

```
val widening : Manager.t -> t -> t -> t
```

Widening

```
val widening_threshold : Manager.t -> t -> t -> Lincons0.t array -> t
```

22.3.7 Closure operation

```
val closure : Manager.t -> t -> t
```

Closure: transform strict constraints into non-strict ones.

```
val closure_with : Manager.t -> t -> unit
```

Side-effect version

```
val print_array :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a array -> unit
```

General use

Part V

Others

Chapter 23

Module Mpz : GMP multi-precision integers

`type t`

GMP multi-precision integers

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like `set`, `add`, ...) the first parameter of type `t` is an out-parameter and holds the result when the function returns. For instance, `add x y z` adds the values of `y` and `z` and stores the result in `x`.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation, unlike the corresponding functions in the module `Mpzf`[2].

23.1 Pretty printing

```
val print : Format.formatter -> t -> unit
```

23.2 Initialization Functions

```
val init : unit -> t
val init2 : int -> t
val realloc2 : t -> int -> unit
```

23.3 Assignement Functions

The first parameter holds the result.

```
val set : t -> t -> unit
val set_si : t -> int -> unit
val set_d : t -> float -> unit
val set_str : t -> string -> int -> bool
val swap : t -> t -> unit
```

23.4 Combined Initialization and Assignment Functions

```
val init_set : t -> t
```

```
val init_set_si : int -> t
val init_set_d : float -> t
val init_set_str : string -> int -> t
```

23.5 Conversion Functions

```
val get_si : t -> nativeint
val get_d : t -> float
val get_d_2exp : t -> float * int
val get_str : int -> t -> string
```

23.6 User Conversions

These functions are additions to or renaming of functions offered by the C library.

```
val to_string : t -> string
val to_float : t -> float
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
```

23.7 Arithmetic Functions

The first parameter holds the result.

```
val add : t -> t -> t -> unit
val add_ui : t -> t -> int -> unit
val sub : t -> t -> t -> unit
val sub_ui : t -> t -> int -> unit
val ui_sub : t -> int -> t -> unit
val mul : t -> t -> t -> unit
val mul_si : t -> t -> int -> unit
val addmul : t -> t -> t -> unit
val addmul_ui : t -> t -> int -> unit
val submul : t -> t -> t -> unit
val submul_ui : t -> t -> int -> unit
val mul_2exp : t -> t -> int -> unit
val neg : t -> t -> unit
val abs : t -> t -> unit
```

23.8 Division Functions

`c` stands for ceiling, `f` for floor, and `t` for truncate (rounds toward 0).

23.8.1 Ceiling division

```
val cdiv_q : t -> t -> t -> unit
```

The first parameter holds the quotient.

```
val cdiv_r : t -> t -> t -> unit
```

The first parameter holds the remainder.

```
val cdiv_qr : t -> t -> t -> t -> unit
```

The two first parameters hold resp. the quotient and the remainder).

```
val cdiv_q_ui : t -> t -> int -> int
```

The first parameter holds the quotient.

```
val cdiv_r_ui : t -> t -> int -> int
```

The first parameter holds the remainder.

```
val cdiv_qr_ui : t -> t -> t -> int -> int
```

The two first parameters hold resp. the quotient and the remainder).

```
val cdiv_ui : t -> int -> int
val cdiv_q_2exp : t -> t -> int -> unit
```

The first parameter holds the quotient.

```
val cdiv_r_2exp : t -> t -> int -> unit
```

The first parameter holds the remainder.

23.8.2 Floor division

```
val fdiv_q : t -> t -> t -> unit
val fdiv_r : t -> t -> t -> unit
val fdiv_qr : t -> t -> t -> t -> unit
val fdiv_q_ui : t -> t -> int -> int
val fdiv_r_ui : t -> t -> int -> int
val fdiv_qr_ui : t -> t -> t -> int -> int
val fdiv_ui : t -> int -> int
val fdiv_q_2exp : t -> t -> int -> unit
val fdiv_r_2exp : t -> t -> int -> unit
```

23.8.3 Truncate division

```
val tdiv_q : t -> t -> t -> unit
val tdiv_r : t -> t -> t -> unit
val tdiv_qr : t -> t -> t -> t -> unit
val tdiv_q_ui : t -> t -> int -> int
val tdiv_r_ui : t -> t -> int -> int
val tdiv_qr_ui : t -> t -> t -> int -> int
val tdiv_ui : t -> int -> int
val tdiv_q_2exp : t -> t -> int -> unit
val tdiv_r_2exp : t -> t -> int -> unit
```

23.8.4 Other division-related functions

```
val gmod : t -> t -> t -> unit
val gmod_ui : t -> t -> int -> int
val divexact : t -> t -> t -> unit
val divexact_ui : t -> t -> int -> unit
val divisible_p : t -> t -> bool
val divisible_ui_p : t -> int -> bool
val divisible_2exp_p : t -> int -> bool
val congruent_p : t -> t -> t -> bool
val congruent_ui_p : t -> int -> int -> bool
val congruent_2exp_p : t -> t -> int -> bool
```

23.9 Exponentiation Functions

```
val powm : t -> t -> t -> t -> unit
val powm_ui : t -> t -> int -> t -> unit
val pow_ui : t -> t -> int -> unit
val ui_pow_ui : t -> int -> int -> unit
```

23.10 Root Extraction Functions

```
val root : t -> t -> int -> bool
val sqrt : t -> t -> unit
val sqrtrem : t -> t -> t -> unit
val perfect_power_p : t -> bool
val perfect_square_p : t -> bool
```

23.11 Number Theoretic Functions

```
val probab_prime_p : t -> int -> int
val nextprime : t -> t -> unit
val gcd : t -> t -> t -> unit
val gcd_ui : t option -> t -> int -> int
val gcdext : t -> t -> t -> t -> t -> unit
val lcm : t -> t -> t -> unit
val lcm_ui : t -> t -> int -> unit
val invert : t -> t -> t -> bool
val jacobi : t -> t -> int
val legendre : t -> t -> int
val kronecker : t -> t -> int
val kronecker_si : t -> int -> int
val si_kronecker : int -> t -> int
val remove : t -> t -> t -> int
val fac_ui : t -> int -> unit
val bin_ui : t -> t -> int -> unit
```

```
val bin_uiui : t -> int -> int -> unit
val fib_ui : t -> int -> unit
val fib2_ui : t -> t -> int -> unit
val lucnum_ui : t -> int -> unit
val lucnum2_ui : t -> t -> int -> unit
```

23.12 Comparison Functions

```
val cmp : t -> t -> int
val cmp_d : t -> float -> int
val cmp_si : t -> int -> int
val cmpabs : t -> t -> int
val cmpabs_d : t -> float -> int
val cmpabs_ui : t -> int -> int
val sgn : t -> int
```

23.13 Logical and Bit Manipulation Functions

```
val gand : t -> t -> t -> unit
val ior : t -> t -> t -> unit
val xor : t -> t -> t -> unit
val com : t -> t -> unit
val popcount : t -> int
val hamdist : t -> t -> int
val scan0 : t -> int -> int
val scan1 : t -> int -> int
val setbit : t -> int -> unit
val clrbit : t -> int -> unit
val tstbit : t -> int -> bool
```

23.14 Input and Output Functions: not interfaced

23.15 Random Number Functions: see Gmp_random[25] module

23.16 Integer Import and Export Functions

```
val import :
  t ->
  (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t ->
  int -> int -> unit
val export :
  t ->
  int ->
  int -> int -> (int, Bigarray.int32_elt, Bigarray.c_layout) Bigarray.Array1.t
```

23.17 Miscellaneous Functions

```
val fits_ulong_p : t -> bool
val fits_slong_p : t -> bool
val fits_uint_p : t -> bool
val fits_sint_p : t -> bool
val fits_ushort_p : t -> bool
val fits_sshort_p : t -> bool
val odd_p : t -> bool
val even_p : t -> bool
val size : t -> int
val sizeinbase : t -> int -> int
```

Chapter 24

Module Mpq : GMP multiprecision rationals

type t

GMP multiprecision rationals

The following operations are mapped as much as possible to their C counterpart. In case of imperative functions (like set, add, ...) the first parameter of type t is an out-parameter and holds the result when the function returns. For instance, add x y z adds the values of y and z and stores the result in x.

These functions are as efficient as their C counterpart: they do not imply additional memory allocation, unlike the corresponding functions in the module Mpqf[3].

val canonicalize : t -> unit

24.1 Pretty printing

val print : Format.formatter -> t -> unit

24.2 Initialization and Assignment Functions

val init : unit -> t

val set : t -> t -> unit

val set_z : t -> Mpz.t -> unit

val set_si : t -> int -> int -> unit

val set_str : t -> string -> int -> bool

val swap : t -> t -> unit

24.3 Additional Initialization and Assignements functions

These functions are additions to or renaming of functions offered by the C library.

val init_set : t -> t

val init_set_z : Mpz.t -> t

val init_set_si : int -> int -> t

val init_set_str : string -> int -> t

val init_set_d : float -> t

24.4 Conversion Functions

```
val get_d : t -> float
val set_d : t -> float -> unit
val get_str : int -> t -> string
```

24.5 User Conversions

These functionss are additions to or renaming of functions offeered by the C library.

```
val to_string : t -> string
val to_float : t -> float
val of_string : string -> t
val of_float : float -> t
val of_int : int -> t
val of_frac : int -> int -> t
val of_mpz : Mpz.t -> t
val of_mpz2 : Mpz.t -> Mpz.t -> t
```

24.6 Arithmetic Functions

```
val add : t -> t -> t -> unit
val sub : t -> t -> t -> unit
val mul : t -> t -> t -> unit
val mul_2exp : t -> t -> int -> unit
val div : t -> t -> t -> unit
val div_2exp : t -> t -> int -> unit
val neg : t -> t -> unit
val abs : t -> t -> unit
val inv : t -> t -> unit
```

24.7 Comparison Functions

```
val cmp : t -> t -> int
val cmp_si : t -> int -> int -> int
val sgn : t -> int
val equal : t -> t -> bool
```

24.8 Applying Integer Functions to Rationals

```
val get_num : Mpz.t -> t -> unit
val get_den : Mpz.t -> t -> unit
val set_num : t -> Mpz.t -> unit
val set_den : t -> Mpz.t -> unit
```

24.9 Input and Output Functions: not interfaced

Chapter 25

Module Gmp_random : GMP random generation functions

```
type state
```
GMP random generation functions

25.1 Random State Initialization

```
val init_default : unit -> state
val init_lc_2exp : Mpz.t -> int -> int -> state
val init_lc_2exp_size : int -> state
```

25.2 Random State Seeding

```
val seed : state -> Mpz.t -> unit
val seed_ui : state -> int -> unit
```

25.3 Random Number Functions

```
val urandomb : Mpz.t -> state -> int -> unit
val urandomm : Mpz.t -> state -> Mpz.t -> unit
val rrandomb : Mpz.t -> state -> int -> unit
```